ASSESSMENT COVER SHEET

Student's name	(Surname) Saemo	(Given names) Jeremiah	
ID number	33105081	**Phone**	0411489079
Unit name	Object Oriented Design and Implementation	**Unit code**	FIT2099

Title of assignment	Assignment 1
Is this an authorised group assignment? ☐ **Yes** [✓ ☐ **No**]	
Has any part of this assignment been previously submitted as part of another unit/course? ☐ **Yes** [✓ ☐ **No**]	
Tutorial/laboratory day & time	Friday, 10am - 12pm
Due date: 2024-04-19	**Date submitted**

Intentional plagiarism or collusion amounts to cheating under Part 7 of the Monash University (Council) Regulations

Plagiarism: Plagiarism means taking and using another person's ideas or manner of expressing them and passing them off as one's own. For example, by failing to give appropriate acknowledgement. The material used can be from any source (staff, students or the internet, published and unpublished works).

Collusion: Collusion means unauthorised collaboration with another person on assessable written, oral or practical work and includes paying another person to complete all or part of the work.

Where there are reasonable grounds for believing that intentional plagiarism or collusion has occurred, this will be reported to the Associate Dean (Education) or delegate, who may disallow the work concerned by prohibiting assessment or refer the matter to the Faculty Discipline Panel for a hearing.

Student Statement:

- I have read the university's Student Academic Integrity Policy and Procedures.
- I understand the consequences of engaging in plagiarism and collusion as described in Part 7 of the Monash University (Council) Regulations http://adm.monash.edu/legal/legislation/statutes
- I have taken proper care to safeguard this work and made all reasonable efforts to ensure it could not be copied.
- No part of this assignment has been previously submitted as part of another unit/course.
- I acknowledge and agree that the assessor of this assignment may for the purposes of assessment, reproduce the assignment and:
    1. provide to another member of faculty and any external marker; and/or
    2. submit it to a text matching software; and/or
    3. submit it to a text matching software which may then retain a copy of the assignment on its database for the purpose of future plagiarism checking.
- I certify that I have not plagiarised the work of others or participated in unauthorised collaboration when preparing this assignment.

Signature: Jeremiah Saemo *Date: 2024-04-19*

* delete (iii) if not applicable

Design Rationale

Design Goals for Overall System

- Single Responsibility Principle (SRP): Ensure that design promotes that each class or module have a single responsibility and that all its services are aligned with that responsibility.
- Open/Closed Principle (OCP): Design the game components to be open for extension but closed for modification.
- Liskov Substitution Principle (LSP): Ensure that design of subclasses honours the behavioural contracts established by its super class, and design system in a way that subclasses do not introduce behaviour that contradicts or undermines the expected behaviour of the superclass.
- Interface Segregation Principle (ISP): Ensure that design of classes are not dependent upon interfaces that they do not use, and ensure that interfaces are cohesive, focused, and tailored to the specific needs of the classes that use them.
- Dependency Inversion Principle (DIP): Ensure that design promotes abstraction and reduces coupling between high-level and low-level modules.
- Do Not Repeat Yourself Principle (DRY): Ensure that design eliminates duplication by extracting common code into reusable functions, classes, or modules.

Requirement 1: The Intern of the Static Factory

Brief Summary of Requirements

In the "Static Factory" game, users play as an Intern hired by the Static Factory. The objective of this requirement is to allow the intern to collect scraps from the abandoned moon (the map), which are valuable for the factory's production system.

The primary extension to the system requires that the Intern should have the action of being able to pick up Scrap items and drop them into its spaceship, and that Scrap items should only be picked up and dropped off by the Intern. (See Appendix A for a feature extraction of requirement 1).

Design Goals

In addition to the design goals for the overall system, the design goals for requirement 1 include:

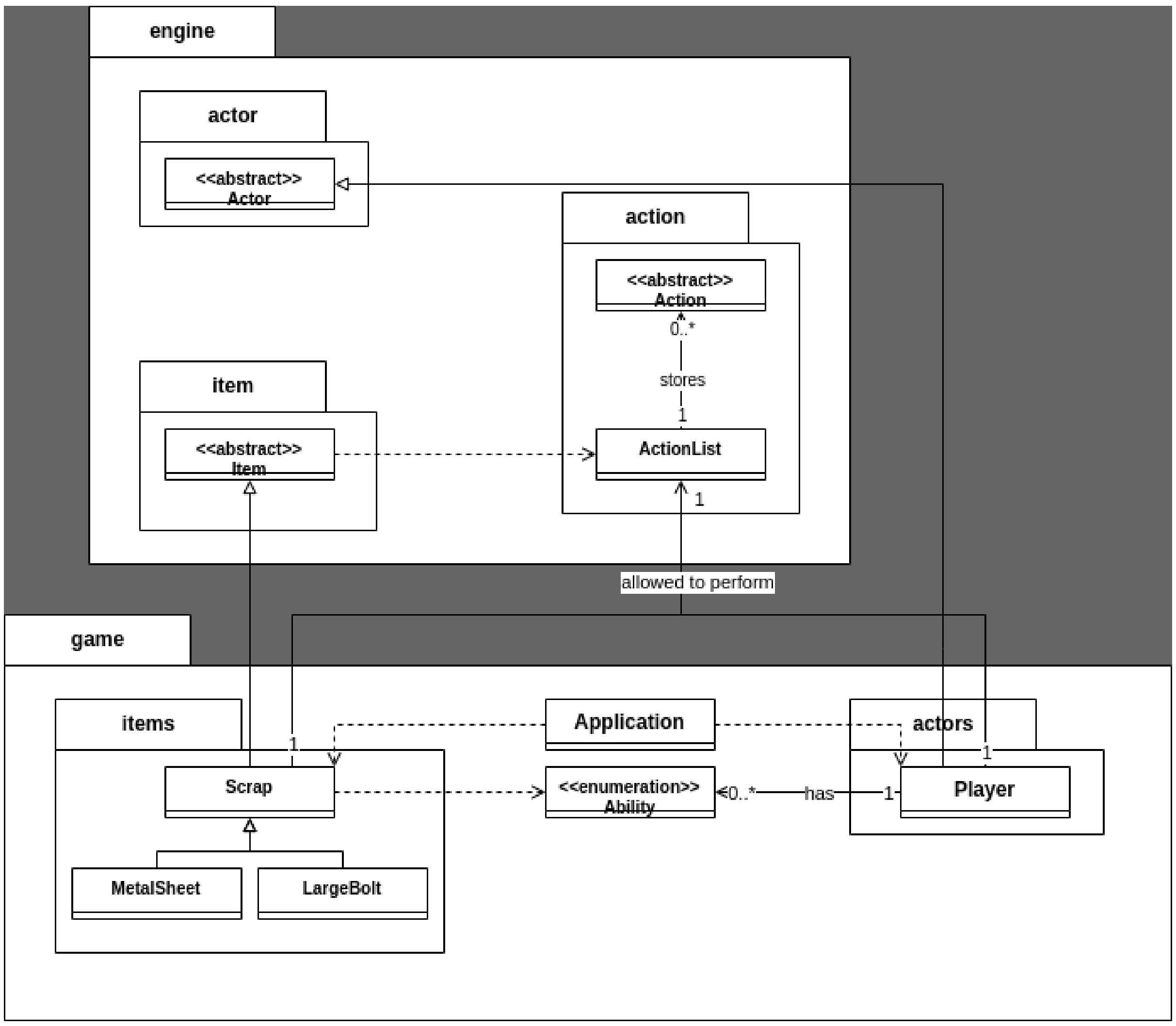
- Create classes from features extracted out of requirement outline
    - Scrap class to extend Item class
    - Player class to extend Actor class
- Ensure that Player is capable of Actions of picking up and dropping off Scrap items
- Place Scrap items onto the Map
- Ensure that Scrap items can only be picked up by the Intern

Overall Concept of Design

The overall concept of the design solution to Requirement 1 is to:

1. Create a Scrap item class by extending from the Item class in the game engine
2. Set Scrap items to not portable by default; that is, that Scrap items cannot be picked up by any Actor, including Player actor
3. Add a constant CAN_PICKUP_SCRAP to the Ability enum
4. Add CAN_PICKUP_SCRAP to the Player's capability set
5. Override allowableActions() methods of the Scrap class, and add a new PickUpAction and DropAction to the Scrap's ActionList if the Actor on its location has the ability CAN_PICKUP_SCRAP

Design


engine
actor
<<abstract>> Actor
action
<<abstract>> Action
0..*
stores
1
item
<<abstract>> Item
ActionList
1
allowed to perform
game
items
Scrap
1
MetalSheet
LargeBolt
Application
<<enumeration>> Ability
0..*
has
1
actors
1
Player

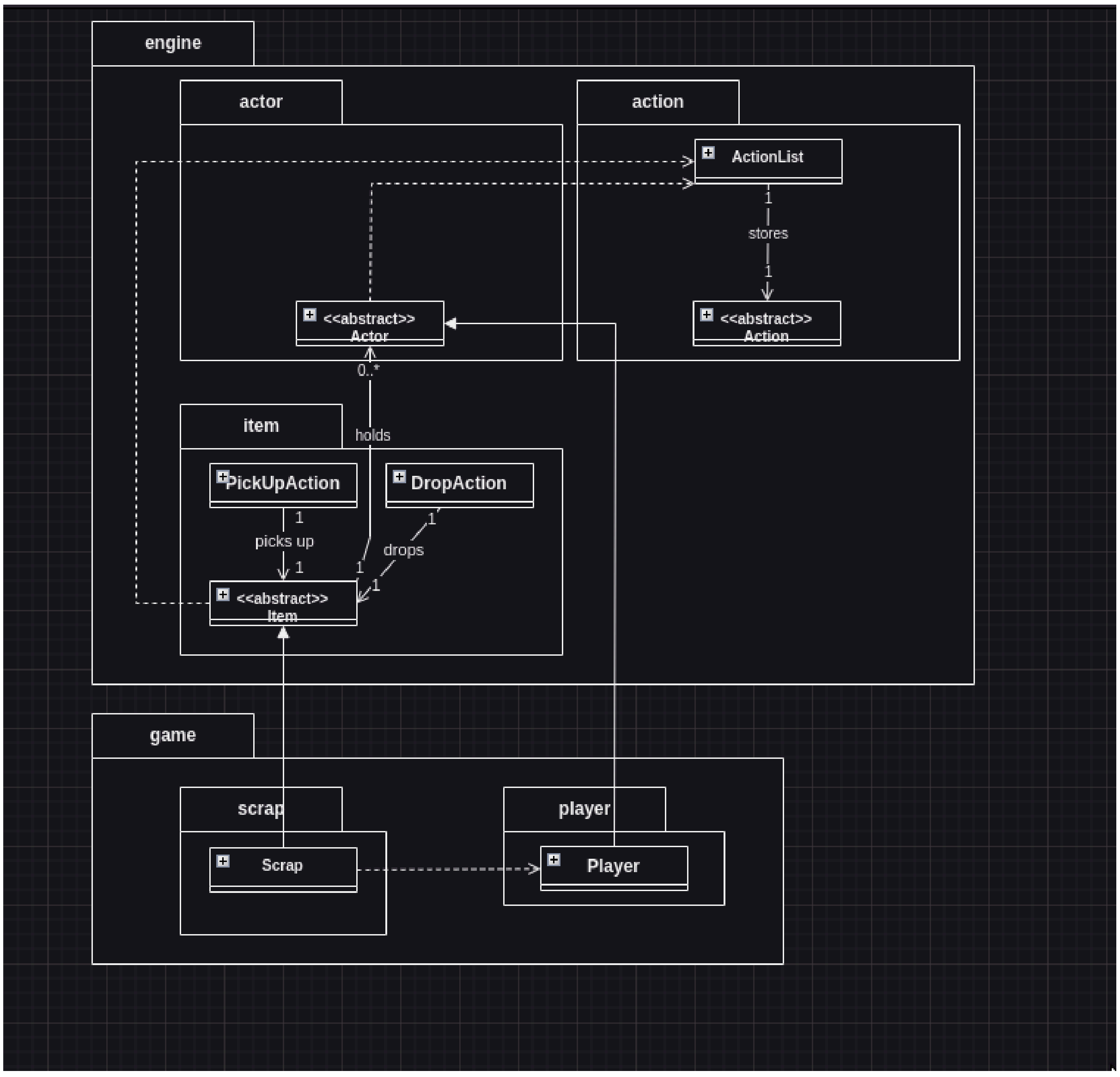
Analysis

The diagram above represents an object-oriented system for requirement 1 of Static Factory, which introduces a Scrap class extending from the Item class, and a Player class extending from the Actor class. Both new classes have associations with the ActionList class, a class which stores Action objects which are allowable to their owners, i.e. Item or Actor. The game package showcases a use of the Ability enumeration that is associated with the Player class and is depended upon by the Item class. The Scrap class is further extended to include data holder classes of MetalSheet and LargeBolt.

One design choice that was made in outlining the blueprint for this requirement was to update the Ability enumeration class with the constant CAN_PICKUP_SCRAP, and to make the Scrap class dependent upon this new constant. In a previous iteration of the design (see previous design below), the Scrap class was designed to depend on the Player class as opposed to the Ability class, by checking if the Actor picking up the Scrap was an instanceof the Player class. This previous design violated the principle of OCP because, if the system were to be extended to allow more Actors the ability to pick up scraps, the Scrap class would need to be modified in order to check the instance of each Actor which has this ability. One of the advantages of the final design is that the system is easily extendable. By designing the requirement in a way where the Scrap class is dependent on the Ability enum as opposed to the Actor who is picking up the Scrap, the Scrap class is closed for modification, meaning all that needs to occur if the system is to be extended to allow more Actors to pick up Scraps is by adding the CAN_PICKUP_SCRAP ability to their capability set.

Another design choice to be mentioned here is the adherence to the SOLID principle of LSP. In a previous design implementation (see previous design below), the solution to requirement 1 was to override the getPickUpAction and getDropAction methods within the Item class, of which would if the passed Actor was an instanceof the Player class. This implementation would violate the principle of Liskov Substitution, as it would introduce behaviour that undermines the expected behaviour of the Item class. The above design implementation, as opposed to overriding getter methods, instead looks to override the allowableActions methods of the Item class.

Previous Design



Requirement 2: The Moon's Flora

Brief Summary of Requirements

Within the surroundings of the ship, the Intern notices a plant growing out of the ground of the moon. This plant can grow and produce fruits, which could be valuable as raw materials for the factory.

Requirement 2 involves extending the game by creating a Plant "Inheritree" that starts as a sapling producing small fruits, then maturing and being able to produce large fruits. These fruits can be picked up and dropped off by the Intern. (See Appendix B for a feature extraction of Requirement 2: The Moon's Flora).

Design Goals

In addition to the design goals for the overall system, the design goals for requirement 2 include:

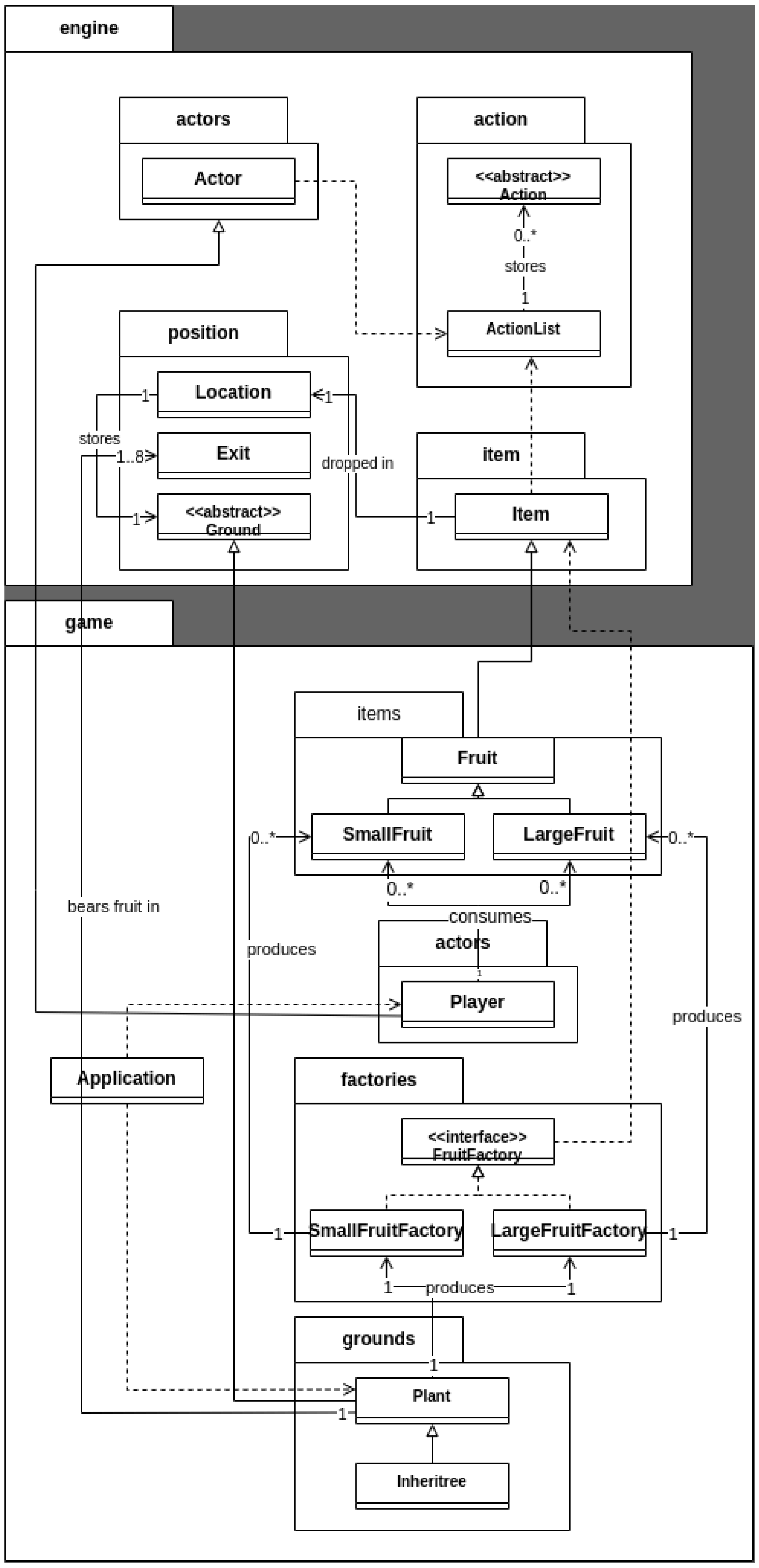
- Create classes from features extracted out of requirement outline
    - Plant class to extend Ground class
    - Fruit class to extend from Item class
    - SmallFruit class to extend from Fruit class
    - LargeFruit class to extend from Fruit class
- Ensure that Plant class handles behaviour of maturing to full grown plant after five ticks
- Ensure that Plant class bears the appropriate fruit according to its maturity
- Ensure that Fruit are borne within the appropriate parameters of their chance of dropping

Overall Concept of Design

The overall concept of the design solution to Requirement 2 is to:

1. Create Plant class by extending Ground class in game engine
2. Create base Fruit class by extending Item class, then extend SmallFruit and LargeFruit from base class
3. Create FruitFactory interface which handles responsibility of generating new Fruit items
4. Implement FruitFactory interface by creating SmallFruitFactory and LargeFruitFactory classes, which are responsible for generating SmallFruit and LargeFruit items
5. Implement SmallFruitFactory and LargeFruitFactory classes within Plant class, so that it is able to bear both small and large fruit
6. Override tick() method of Ground class in order to handle requirement of Plant maturing after five ticks and chance of bearing fruit per tick
7. Build method that generates Fruit items at Exit locations of Plant by calling getExits() method within base Ground class

Design



engine
actors
Actor
action
<<abstract>> Action
0..*
stores
1
ActionList
position
Location
1
<1
stores
1..8>
Exit
dropped in
<<abstract>> Ground
1>
item
Item
1
game
items
Fruit
SmallFruit
LargeFruit
0..*
0..*
0..*
0..*
consumes
actors
Player
1
bears fruit in
produces
produces
Application
factories
<<interface>> FruitFactory
SmallFruitFactory
LargeFruitFactory
1
1
1
produces
1
grounds
1
Plant
1
Inheritree

Analysis

The diagram above represents an object-oriented system for Requirement 2 of Static Factory; a system design that introduces eight new classes into the system:

- Fruit class
- SmallFruit class
- LargeFruit class
- FruitFactory interface
- SmallFruitFactory
- LargeFruitFactory
- Plant class
- Inheritree class

The Fruit class is an extension of the Item class that represents fruit items that are able to be picked up (and consumed). It has been instantiated as a base class from which the classes SmallFruit and LargeFruit extend from. The FruitFactory interface represents a factory for creating new Fruit items, of which is implemented in the SmallFruitFactory and the LargeFruitFactory to create new SmallFruit and LargeFruit items respectively. The Plant class is an extension of the Ground class in the game engine, and relies upon the SmallFruitFactory and LargeFruitFactory classes to bear fruit into its Exits.

One of the design choices made for the fulfilment of Requirement 2 was to create a base Fruit class for SmallFruit and LargeFruit. In further requirements, such as Requirement 4, the SmallFruit and LargeFruit classes were to be extended in order to provide a new HealingAction, which regenerates its owners health by a number of health points. To implement this extended functionality, the Fruit class was created to act as a base from which all fruit classes could extend from. In designing the base class in this way, this design adheres to the DRY principle, since both LargeFruit and SmallFruit both depend on the ability to provide a HealingAction. Furthermore, this design also adheres to the OCP principle, whereby the game may be extended to add more fruits without having to modify the Fruit class, but instead extending from it.

Another design choice that was made for the fulfilment of Requirement 2 was to create a FruitFactory interface, from which would be implemented in factory classes for SmallFruit and LargeFruit. The design of the FruitFactory interface was based on the Interface Segregation Principle in that the FruitFactory interface outlines only one method that its dependents need to implement: the createFruit() method. Because factories are responsible for generating new objects, designing the FruitFactory interface in this way means that the FruitFactory clients are not forced to depend on interfaces that they do not use, and are not imposed additional methods that they do not need.

In a previous iteration of the design (see previous design below), the Plant class relied on a Spawn class, which handled the responsibility of spawning items onto the map. This Spawn class would also be responsible for spawning actors onto the map, as outlined by the requirements of Requirement 3 of spawning HuntsmanSpiders from the Crater. This design presented the problem of implementing a God class, where the Spawn class was designed to handle the responsibility of spawning all items and actors onto the map. God classes are a code smell which show an unfair distribution of functions within a system, and can very easily become unmaintainable. By creating factories for SmallFruit and LargeFruit, I was able to modularise my design and thus adhere to the principle of SRP, where each class is assigned a single responsibility. Here, each factory is responsible for creating a specific type of fruit: SmallFruit or LargeFruit.

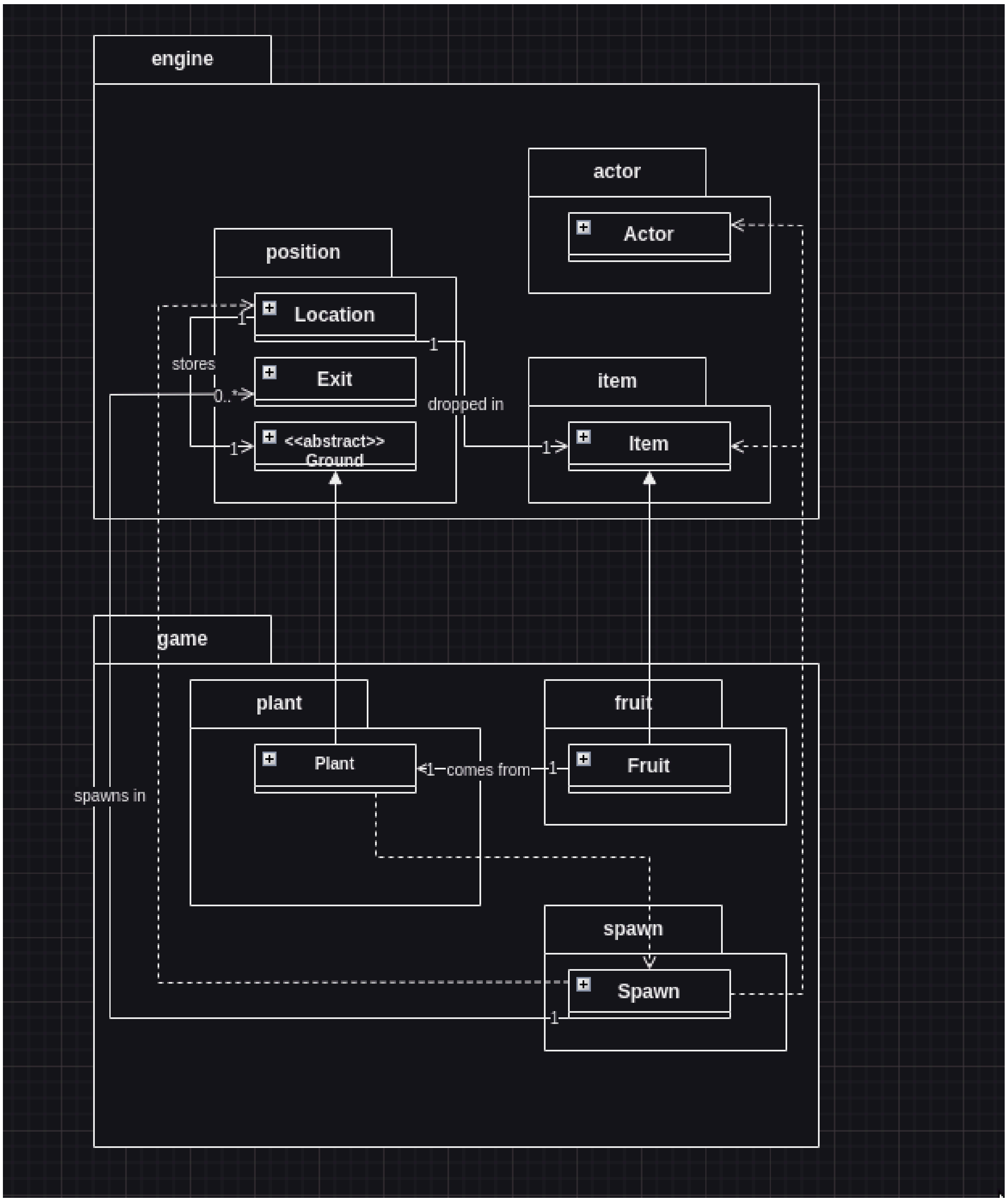
Also within a previous iteration of the design, the Plant class was designed to handle the responsibility of bearing Fruit objects. The Plant class was also designed to be a God class in the sense that it would be responsible for 'growing' Fruit by updating its displayChar to uppercase. Each Fruit 'object' that would be borne from a Plant, if it was implemented in this way, would essentially be a copy of the same object onto the map, as Fruit would be an association of Plant by being one of its attributes.

The previous design also presents another disadvantage: the Plant class and a dependency on a single Fruit class. In the previous design, the Plant class was designed to handle the responsibility of creating new Fruit objects. The previous design had no classes for SmallFruit or LargeFruit, but was instead designed in a way where the Plant class required Fruit objects to be an attribute of the Plant class. This meant that the design solution to growing fruit objects was to implement a grow function within the Plant class which simply updated the displayChar of the Fruit from 'o' to 'O'. This previous design clearly violated the principle of SRP.

In designing the Plant class in this way, the Plant class also became a God class, since it was responsible for 'growing' the Fruit and updating the displayChar of the Fruit. By creating classes for SmallFruit and LargeFruit, the design solution became much simpler, since the Plant class was no longer responsible for updating a single Fruit class to be both a "small" and a "large" fruit.

By creating a design which is based around factories, the Plant class adheres to the Open/Closed principle, since it does not need to be modified if more Fruit were to be added to the game in the future. The implementation of factories, as outlined in the above design, would mean that new Fruit objects would be created, as opposed to simply recalling the same object within a Plant's set of fields, and thus creating errors logic errors in the future if these objects were to be removed from a Player's inventory, such as when being consumed.

Previous Design



Requirement 3: The Moon’s (hostile) Fauna

Brief Summary of Requirements

Requirement 3 involves extending the system by incorporating a crater class that is responsible for spawning Huntsman Spiders, with a chance of spawning this creature of 5% per turn. The HuntsmanSpider should be able to attack the Intern with 25% accuracy if it is in its vicinity - or one Exit away from it. (See Appendix C for a feature extraction of Requirement 3: The Moon’s (hostile) Fauna).

Design Goals

In addition to the design goals for the overall system, the design goals for requirement 3 include:

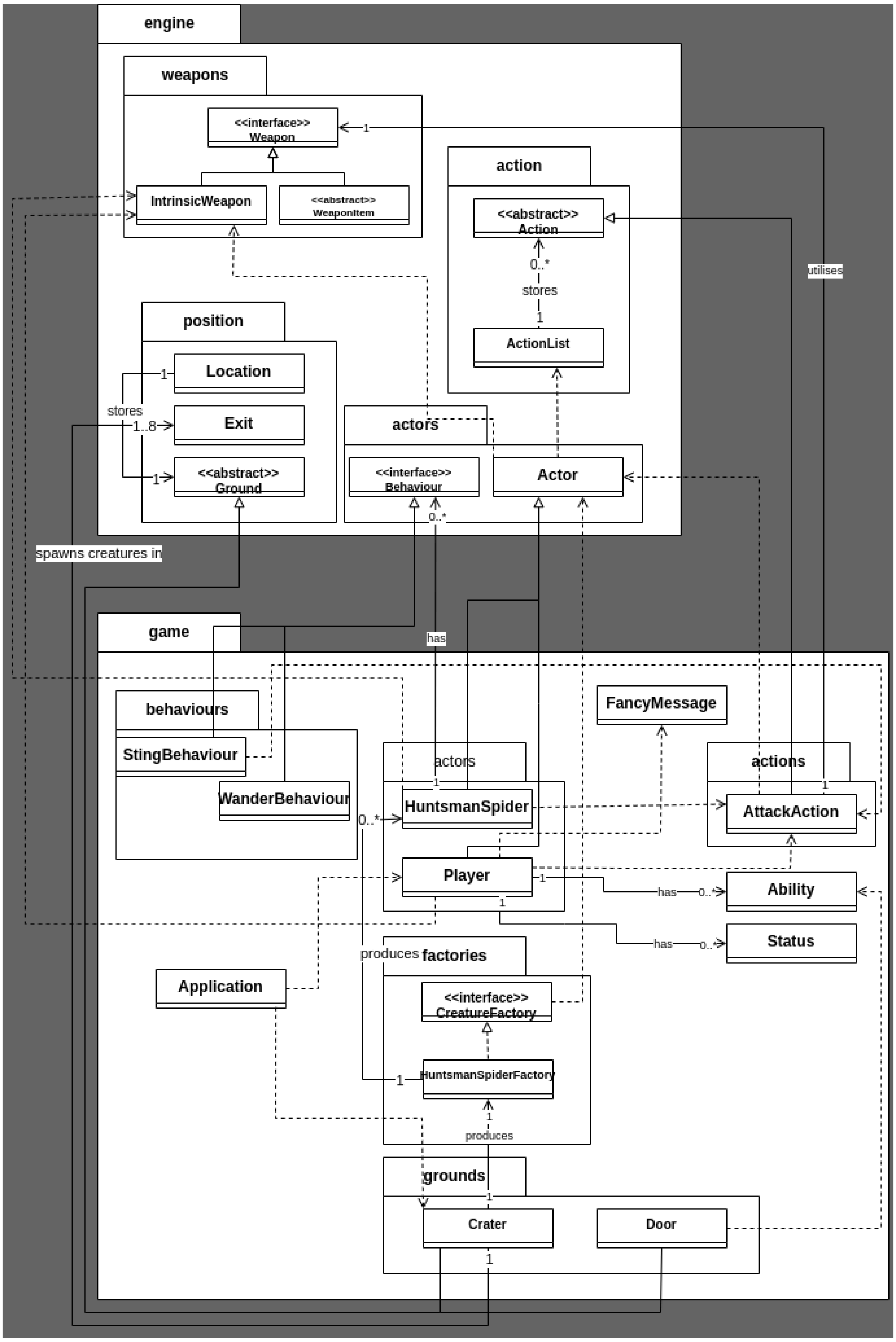
- Create/extend classes from features extracted out of requirement outline
    - Crater class to extend Ground class
    - HuntsmanSpider class to extend from Actor class
- Ensure that implementation handles behaviour of spawning new HuntsmanSpider objects
- Ensure that HuntsmanSpider is able to attack Actors with HOSTILE_TO_ENEMY in their capabilities
- Ensure that HuntsmanSpider classes cannot enter the spaceship of the Intern
- Ensure that Player class and HuntsmanSpider class both have access to an IntrinsicWeapon

Overall Concept of Design

The overall concept of the design solution to Requirement 3 is to:

1. Create Crater class by extending Ground class in game engine
2. Extend HuntsmanSpider class and Player class to give them an IntrinsicWeapon
    a. Assign HuntsmanSpider IntrinsicWeapon to deal 1 damage with 25% accuracy
3. Create CreatureFactory interface, which handles responsibility of generating new creature Actors
4. Implement CreatureFactory interface by creating HuntsmanSpiderFactory, which is responsible for generating new HuntsmanSpider creature Actors
5. Implement HuntsmanSpider classes within Crater class, so that the Crater is able to spawn HuntsmanSpider objects
6. Override tick() method of Ground class within Crater class in order to handle requirement of spawning a new HuntsmanSpider at a chance per tick
7. Build method that spawns HuntsmanSpider actors at Exit locations of Crater by calling getExits() method within base Ground class
8. Create StingBehaviour by extending Behaviour class that returns new AttackAction, and add it to HuntsmanSpider behaviours map
9. Display FancyMessage if Player reaches 0 hit points
10. Create Door class that extends Ground class, and ensure that it can only be entered by Actors who are allowed to enter the spaceship

Design



engine
weapons
<<interface>> Weapon
IntrinsicWeapon
<<abstract>> WeaponItem
action
<<abstract>> Action
0..*
stores
1
ActionList
position
Location
Exit
<<abstract>> Ground
stores
1..8
actors
<<interface>> Behaviour
Actor
0..*
spawns creatures in
utilises
has
game
behaviours
StingBehaviour
WanderBehaviour
actors
HuntsmanSpider
Player
FancyMessage
actions
AttackAction
Ability
Status
has
0..*
Application
produces
factories
<<interface>> CreatureFactory
HuntsmanSpiderFactory
produces
grounds
Crater
Door

Analysis

The diagram above represents an object-oriented system for Requirement 3 of Static Factory; a system design that extends the system by introducing five new classes into the system:

- Crater
- CreatureFactory
- HuntsmanSpiderFactory
- StingBehaviour
- Door

The design above has been made with the intention of extending the following existing classes:

- Ability
- HuntsmanSpider
- Player

The Crater class is similar to the Plant class, except that it is responsible for generating HuntsmanSpiders in its Exit locations as opposed to Fruit. The same design choices that were made in designing the implementation for REQ-2 have been made in REQ-3, in the sense that the design relies on factory classes in order to meet the requirements of spawning HuntsmanSpider objects. The CreatureFactory Interface represents a factory for creating new creature-type Actors and outlines only one method for extended classes to implement. The HuntsmanSpiderFactory class is responsible for implementing the createCreature() function, which is designed to return new HuntsmanSpider objects. The design of these factory classes adheres to the SOLID principle of ISP, as CreatureFactory clients are not forced to depend on interfaces that they do not use, and are not imposed additional methods that they do not need.

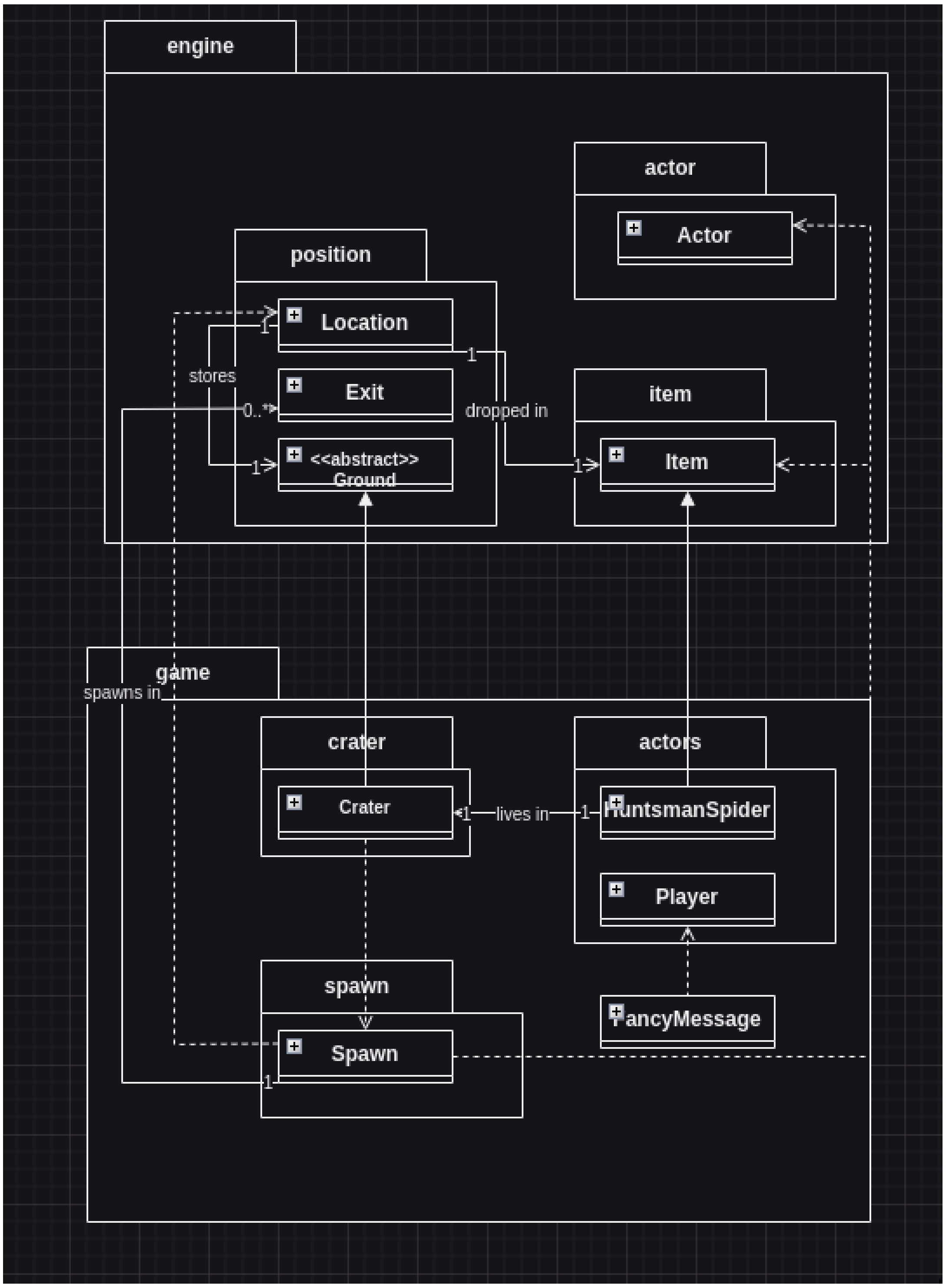
In a previous iteration of the design (see Previous Design 1 below), the Crater class relied on a Spawn class, which handled the responsibility of spawning Actors onto the map. This design presented the problem of potentially implementing a God class, whereby the Spawn class would handle the responsibility of spawning all Actor objects, as well as all Item objects, within the game. This design broke the SRP principle, and as such was reiterated to the design solution above. The Single-Responsibility Principle is adhered to in the above design as the factory classes modularise the Spawn class into single-responsibility classes, which is to create a new Item (createFruit() in FruitFactory), or to create a new Actor (createCreature() in CreatureFactory). By creating a CreatureFactory interface, the design also adheres to the OCP principle, because, if the game were to be extended in the future in order to add new creatures to the game, then the game would be able to be extended by simply creating a new creature class and extending the CreatureFactory interface.

The StingBehaviour class is introduced in Requirement 3 as an extension of the Behaviour class. It is designed for the HuntsmanSpider class, and is responsible for implementing the requirement of HuntsmanSpiders having the ability to attack the intern. In a previous design iteration of this requirement (see Previous Design 2 below), the ability of HuntsmanSpiders attacking the Intern was implemented by modifying existing code within the HuntsmanSpider class, which clearly violates the Open-Closed principle.

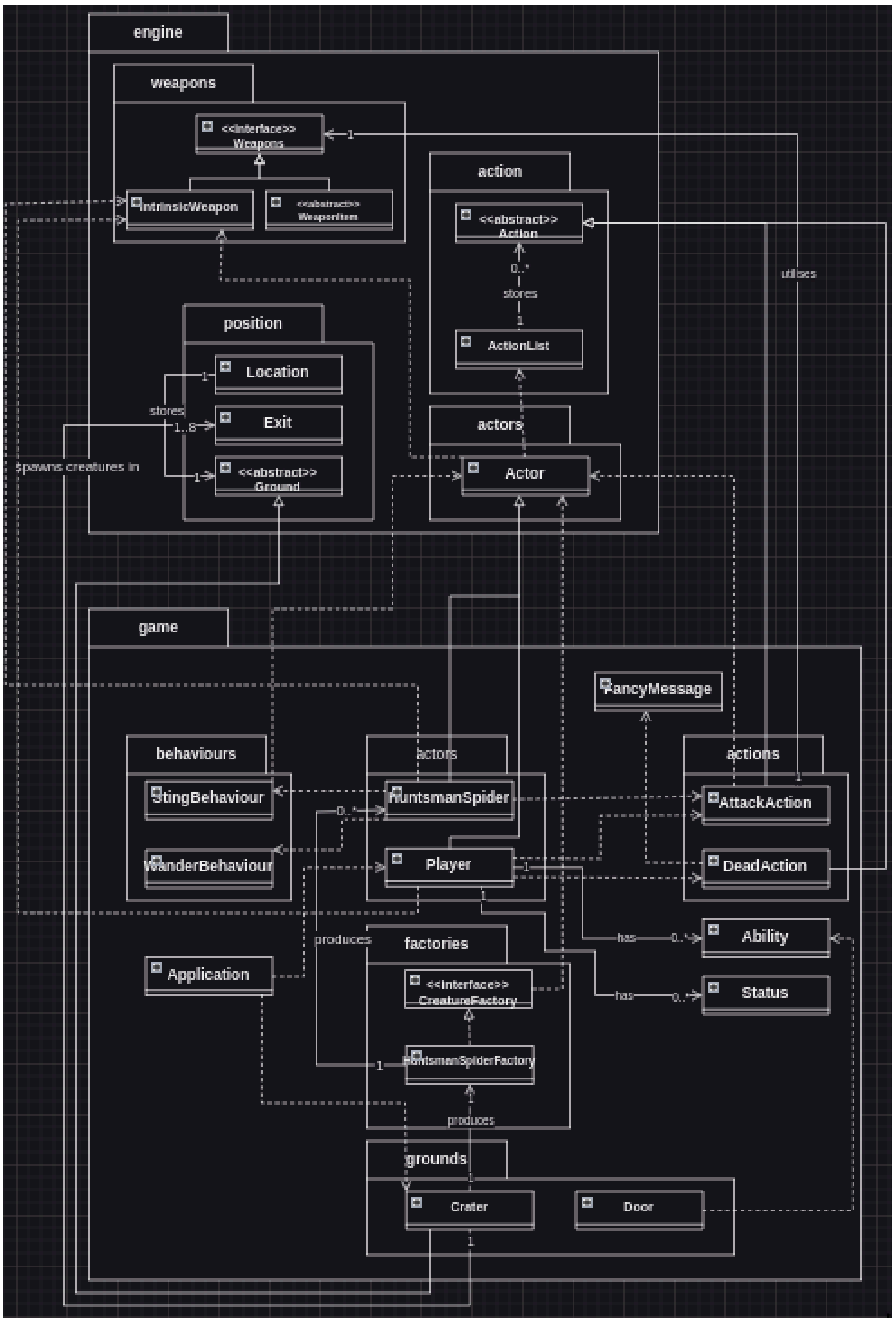
The above design of the StingBehaviour class defines a new implementation of the Behaviour interface by returning a new AttackAction in its overridden getAction() method. Through the above design of the attacking requirement, the implementation adheres to both the OCP and LSP, since no existing code has been modified in order to extend the system, and that subclasses have been designed in such a way that does not introduce behaviour that contradicts or undermines the expected behaviour of the superclass.

The above design outlines a dependency of the Player class on the FancyMessage class in order to meet the requirement of printing a fancy message if the Player became unconscious. In a previous design iteration (see Previous Design 2 below), this requirement was designed in such a way to create a DeadAction, of which was responsible for handling the logic of this requirement. Since no other Actor would ever be allowable to take this Action (since the game does not end when other actors die), the design of DeadAction was a redundant solution to the requirement. In the above design, the Player class is dependent on the FancyMessage class, as opposed to DeadAction being dependent on it. The above design adheres to the principle of SRP, since the Player class is designed in a way to handle its own state of unconsciousness, and not dependent on another class to handle this responsibility.

Previous Design 1



Previous Design 2



Requirement 4: Special Scraps

Brief Summary of Requirements

After spending time looking for scraps, the intern finds some SpecialScraps such as MetalPipes that could help them survive on the abandoned moon. Requirement 3 involves extending the game by creating two SpecialScrap classes: the MetalPipe, which the Intern can interact with to attack hostile creatures, and the Fruit classes, which the Intern can interact with to consume and heal by a number of hit points. (See Appendix D for a feature extraction of Requirement 4: Special Scraps).

Design Goals

In addition to the design goals for the overall system, the design goals for requirement 4 include:

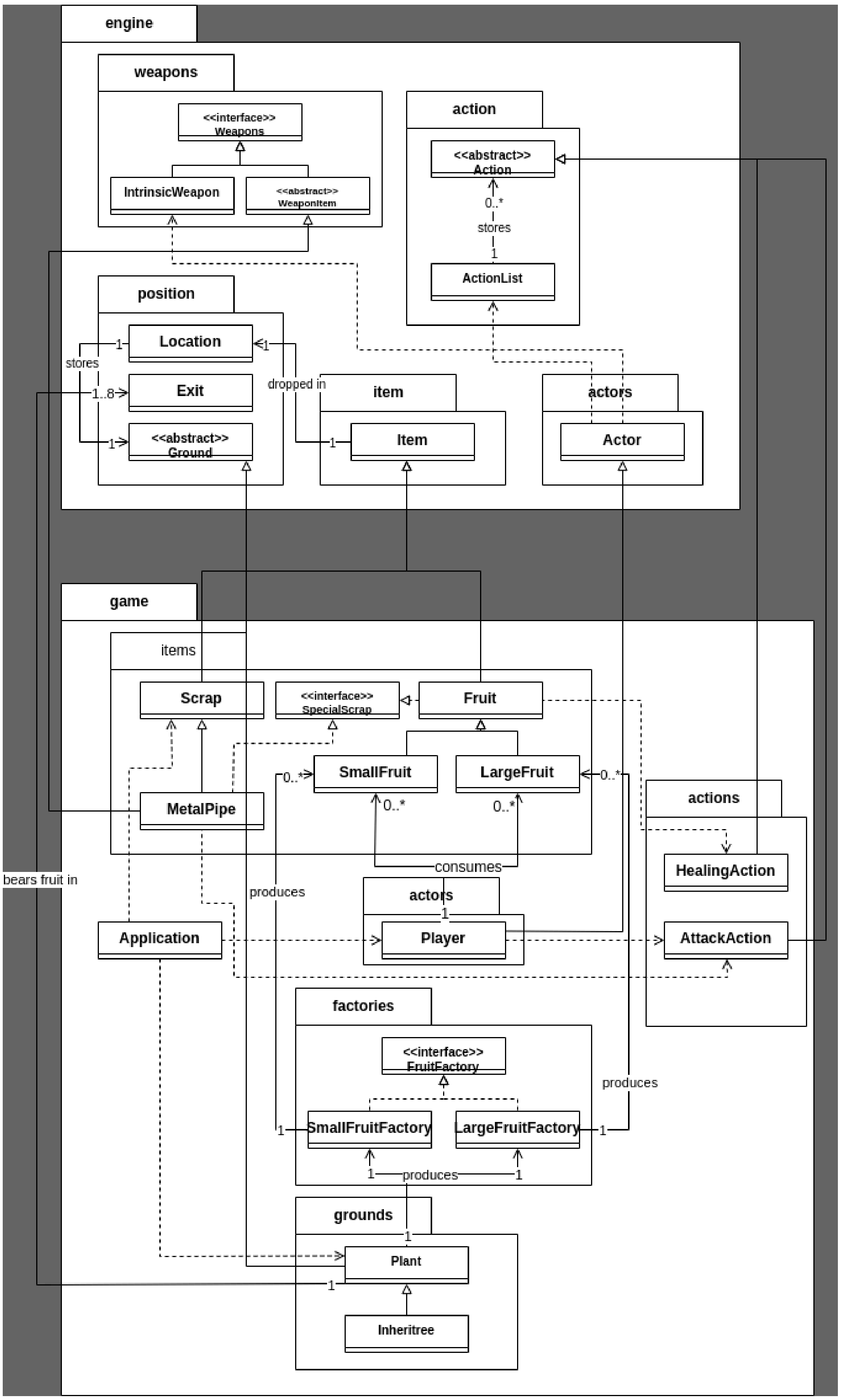
- Create/extend classes from features extracted out of requirement outline
    - MetalPipe class to extend from WeaponItem class
    - Extend Player class from Actor class to override getIntrinsicWeapon()
    - HealingAction class to extend from Action class
    - AttackAction class to extend from Action class
    - SpecialScrap Interface
- Ensure that implementation allows Intern to attack with both IntrinsicWeapon and MetalPipe when MetalPipe is in Intern's inventory
- Ensure that Fruit items heal Intern by correct number of hit points
- Ensure that SpecialScrap interface outlines only the necessary behaviours for child classes to implement

Overall Concept of Design

The overall concept of the design solution to Requirement 4 is to:

1. Create new HealingAction class responsible for an Item, responsible for healing its owner by specified number of health points
2. Extend Fruit class by adding HealingAction in its allowableActions list
3. Create MetalPipe class by extending WeaponItem
4. Extend MetalPipe class by adding AttackAction in its allowableActions list
5. Create SpecialScrap interface for future implementation

Design


engine
weapons
<<interface>> Weapons
IntrinsicWeapon
<<abstract>> WeaponItem
action
<<abstract>> Action
0..*
stores
1
ActionList
position
Location
Exit
<<abstract>> Ground
stores
1
1..8
1
dropped in
item
Item
actors
Actor
game
items
Scrap
<<interface>> SpecialScrap
Fruit
SmallFruit
LargeFruit
MetalPipe
0..*
consumes
produces
bears fruit in
actions
HealingAction
AttackAction
Application
actors
Player
factories
<<interface>> FruitFactory
SmallFruitFactory
LargeFruitFactory
produces
grounds
Plant
Inheritree

Analysis

The diagram above represents an object-oriented system for Requirement 4 of Static Factory; a system design that extends the system by introducing three new classes into the system:

- SpecialScrap interface
- MetalPipe class
- HealingAction class

The design above has been made with the intention of extending the following existing classes:

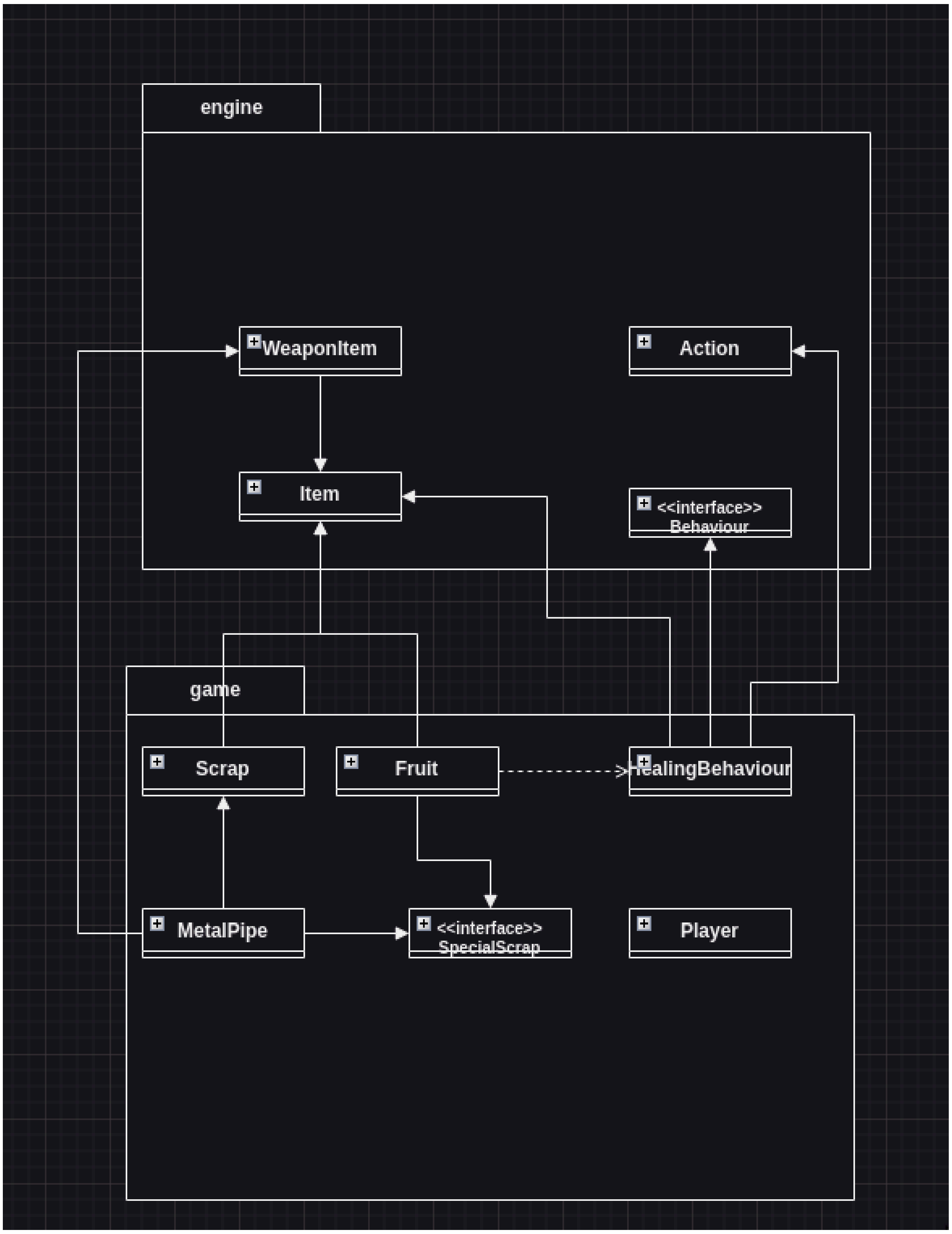
- Fruit class

The above design has been brought about mainly through the concept of abstraction and extending existing classes. When creating the MetalPipe class, a design decision was made to implement the WeaponItem class, as the WeaponItem class had an already established behavioural contract for the behaviours necessary for the MetalPipe class. Another approach to this requirement could be to implement MetalPipe as an Item class and to implement the behaviours of the Weapon interface, but such a design solution would violate the principle of Liskov Substitution. In designing the system in the way above, the implementation of the requirement honours the behavioural contracts established by its super class, and the system would be designed in a way where the MetalPipe does not introduce behaviour that contradicts or undermines the expected behaviour of the superclass.

The SpecialScrap interface defines the body of methods necessary for item classes to implement in order to be considered a SpecialScrap. In this requirement, two classes are identified as being SpecialScraps: the MetalPipe and the Fruit classes. The above design showcases that MetalPipe is dependent on an AttackAction, and that Fruit are dependent upon a HealingAction. These two classes provide these Actions through inheriting the allowableActions() methods from their superclass: the Item class. Being that the MetalPipe class and the Fruit class provide the functionality of allowing an Action to their owner without being dependent on the SpecialScrap interface, the interface does not have to outline a behaviour of providing an Action for each SpecialScrap class, and thus has been designed in such a way where the interface simply is added for future sprints of the project. Here, it is known that SpecialScraps will be able to be traded in future iterations of the project.

The SpecialScrap interface has been designed in such a way that classes do not have to rely on behaviours that it doesn’t need to implement. By ensuring that MetalPipe and Fruit do not have to rely on implementing behaviours of the SpecialScrap class if they do not need it, the design adheres to the principle of Interface Segregation. The SpecialScrap interface has been focused and tailored to the specific needs of the classes that use them; that being that SpecialScraps will be extended to be bought and sold in the future. Because this behaviour is beyond the scope of the sprint, it has been left to be extended in future sprints and the interface is left empty.

Previous Design



Appendix

Appendix A: Feature extraction for Requirement 1

Feature Extraction

Yellow = classes
Green = attributes/behaviours

Requirement 1: The Intern of the Static factory

Story

In “Static Factory”, you are playing as an Intern hired by the Static factory. Your objective is to collect scraps from abandoned moons, which are valuable for the factory’s production system. You are sent to one of the abandoned moons, “Polymorphia”, for your first mission.

Requirement

The player (@) starts the mission inside their spaceship. They begin with 4 hit points (the health attribute).

The spaceship is represented by several squares of floor (_) surrounded by walls (#) in the middle of the game map. The intern can walk around the floor of the spaceship, but cannot step on the walls.

There are scraps scattered across the map that the Intern needs to pick up: large bolts (+) and metal sheets (%). The Intern has an almost unlimited size inventory to pick up a lot of scraps.

In this requirement (REQ1), these scraps can ONLY be picked up and dropped off by the Intern.

Once the Intern has picked up these scraps, the goal is to drop them inside their spaceship.

Appendix B: Feature extraction for Requirement 2

Requirement 2: The moon's flora

Story

Within the surroundings of the ship, the Intern notices a plant growing out of the ground of the moon. Scanning the plant shows that it is named "Inheritree" and can grow and produce fruits, which could be valuable as raw materials for the factory.

Requirement

The "Inheritree" plant starts as a sapling (t) when it first appears on the moon. At this stage, it can produce small fruits (o), which have a 30% chance of dropping within the plant's surroundings (i.e. one "exit" away, next to the tree), which the Intern can pick up or drop off.

After 5 ticks, the plant can grow to its mature stage (T). Once matured, the plant can produce larger fruits (O) that have a 20% chance of dropping within the plant's surroundings (again, one "exit" away). Like the smaller ones, the Intern can pick up the larger fruits or drop them off

Appendix C: Feature extraction for Requirement 3

Requirement 3: The moon's (hostile) fauna

Story

It turns out the Intern has a new objective: survive.

Requirement

The moon has several craters (u) that can spawn large and hostile Huntsman spiders (8). Each crater can spawn this creature with a 5% chance at every game turn.

This creature will wander around if the Intern is not within it surroundings, (i.e. one "exit" away). However, when the Intern enters the creature's surroundings, the creature will attack the Intern with one of its long legs, dealing 1 damage with 25% accuracy. It cannot attack any other creatures that are hostile to the Intern. The creature has 1 hit point.

NOTE: Don't forget to display the player's hit point (the health attribute of the player) on the screen

NOTE: If the player's hit point reaches 0, make sure to display a message on the screen. You can use the message provided in the FancyMessage class.

The creature cannot enter the Intern's spaceship.

- Consider creating a Door class that extends from Ground in which can only be accessed if Actor is instanceof Player

Appendix D: Feature extraction for Requirement 4

Requirement 4: Special scraps

Story

After spending several days looking for scraps for the factory, the intern finds some useful scraps that could help them survive on the abandoned moon, such as metal pipes. But, of course, meeting the factory's scrap quota is the highest priority…

Requirement

The metal pipe (!) is one type of a special scrap that the intern can interact with. If it is in the Intern's inventory, they can use it to attack hostile creatures on the moon, dealing 1 damage with 20% accuracy. Similar to other scraps, the metal pipe can be picked up or dropped off.

The Intern can also punch hostile creatures with their bare hands, dealing 1 damage with 5% accuracy.

NOTE: The player should be given the option to punch the hostile creatures regardless of whether they are carrying a metal pipe. In other words, if the player carries a metal pipe, they are given two options to attack the hostile creatures: with their bare hands or the metal pipe.

The Intern can only attack a hostile creature if the hostile creature is within the Intern's surroundings.

Next, the fruits produced by the Inheritree plant can be consumed by the player. The smaller fruits can heal the player by 1 hit points, while the larger ones can heal the player by 2 hit points. Once consumed, the fruit should be removed from the Intern's inventory. These fruits are also special scraps.

NOTE: what if there are more special scraps? Can you implement these new scraps easily, given your current design? These scraps would be traded (bought/sold) in the future assignment.